DESIGN RATIONALE: FIT2099 SSB ASSIGNMENT 3

GROUP NAME: JAVAPLUSPLUS

GROUP MEMBERS: MOHAMED SHAKEEL MOHAMED RAFI 28021452

MATTI HADDAD 29708966

KERRY YUE SONG ZHENG 28794346

This document relays the design rationale and thought process for the tasks that were assigned.

1. Leave Affordance

Because none of the other classes can perform this type of affordance and each actor can perform the 'Leave action', having it as a subclass of HpAffordance will increase reusability, maintainability and reduce dependency as it will perform the action in its own module.

If an actor is carrying items (ring, wand, sword), the actor has the option of 'leaving' an item in the actor's current location. Each item 'left' consumes one turn.

The Leave class is a subclass of HPAffordance that depends on:

- MessageRender to display a message to the user, a form of player feedback.
- HPEntetityInterface as it deals with entities and therefore the item needs to be managed by EntityManager. An HP item would not exist without the other.
- HPActor as actors are the ones who initiate the action.
- HPAction as to check if the action can be performed (an actor cannot give an item to an actor holding an item).

2. Give Affordance

Just as leave, each actor can perform the give action, having it as a subclass of HpAffordance will increase reusability, maintainability and reduce dependency as it will perform the action in its own module.

If an actor is carrying items (ring, wand, sword), the actor has the option of 'giving' an item to another actor if and only if both actors are in the same location and both are from the same team. In addition, the item can be rejected by the receiver, if so, it will stay with the actor that tried to 'give' it, each item 'given' consumes one turn whether the item was accepted or declined.

The Give class is a subclass of HPAffordance that depends on:

- MessageRender to display a message to the user, a form of player feedback.
- HPEntetityInterface as it deals with entities and therefore the item needs to be managed by EntityManager.
- HPActor as actors are the ones who initiate the action.
- HPAction as to check if the action can be performed (an actor cannot give an item to an actor holding an item).

3. Wand and Spell Implementation

For a Spell to be Casted, there must be a Wand class and a Cast Action. Furthermore, Cast may target both entities and actors, which will require further distinction.

3.1 Wand and Cast Class

A Wand, like a dagger or any other item in the game, can be picked up by any Actor. The existence of a Wand allows an Actor to Cast Spells onto other Actors, or itself. The wand will have the capability casting so that it is easy to determine the function of the wand, which is also in line with many of the existing entities such as sword and axes.

3.2 Cast and Actor's Known Spells

Casting a spell requires a new action - Cast. This is different from Attack as it requires a Spell to be an input as well.

It will then execute the Spell's effect if the Actor knows the Spell.

Every actor will have a set of its known Spells.

The reason cast is not a subclass of Attack as it was previously proposed in the previous assignment because of a need of abstraction between casting and each individual spell. Different spells can be grouped together but it cannot all be a subclass of Cast because some spells require different parameters that cast cannot possibly offer without creating excessive dependencies and extensive amounts of parameters. Cast was not created as a subclass of attack because the requirements to attack were more in line with other affordances so it was made a subclass of HPAffordance.

The enum spells was also removed because there was no need for it, since all spells are subclasses of Spells, making it redundant.

3.3 Targeting Items

To allow Spells to be Casted on items, the interface HPEntityInterface is used in Cast and Spell to target both subclasses.

Additional checks must be implemented for individual spells to ensure that the target is the intended class type.

3.4 Expelliarmus

How expelliarmus works is that by casting this spell on a valid target i.e. a target that is not on the same team as the caster, by "forcing" the targeted actor to drop the item they are holding if they are holding one. The leave class would first check if the targeted actor is holding an item. If they are, it would remove the item the actor is carrying, by changing itemCarried attributes of HPActor, and place it in the location the targeted actor is at. As such, expelliarmus is dependent on leave. By using the leave class, it makes use of the existing affordance leave, and the functionality of expelliarmus is essentially an actor dropping an item (leave affordance) involuntarily.

3.5 Immobulus

The spell Immobulus when cast on an actor would render them unable to perform any actions such as moving, attacking or performing any affordances such as dropping or taking an item, essentially skipping one of their turns. In order to implement this function, we would ignore an actor's input to perform any actions for the duration of the spell.

3.6 Avada Kedavra

The spell Avada Kedavra works by casting it on an actor and killing them instantly. In order to this, we would need access the targeted actor's health and make it so that the targeted actor takes an amount of damage equal to the current health, killing them instantly, so it would be dependent on HPActor. Additionally, because dead actor's cannot hold weapons, it would also drop any items the targeted actor was holding by making use of the Leave class.

4. Potions

If an increaseHitPoints method is added to HPEntity, health potions could be implemented as class instances of HPEntity. But since potions are more interesting, e.g. magic, boost attack potions, it was decided that it should be a subclass of HPEntity, it uses some of its functionality, in addition it will have a method increaseHealth.

Potions are used to allow an actor to replenish a random number of hit points if and only if the actor's health is not full. Potions cannot increase the actor's health beyond its default health. Furthermore, Potions must be 'taken' first, then the Drink affordance is used to 'drink' the potions, this consumes the actor's turn. Potion also have an initial position, can be visible or hidden.

5. Drink

A subclass of HPAffordance, this follows the same design as other affordances.

If an actor is holding an item with a HEALTH capability, then the actor has the option of 'drinking' the item.

The Drink class is a subclass of HPAffordance that depends on:

- MessageRender to display a message to the user, a form of player feedback.
- HPEntetityInterface as it deals with entities and therefore the item needs to be managed by EntityManager.
- HPActor, as actors are the ones who initiate the action.
- HPAction, as to check if the action can be performed.

6. Cast

Because cast works differently from attack and it fairly similar to actions which are subclasses of HPAffordance, it is therefore appropriate for Cast to also be a subclass of HPAffordance

7. Spells

Because all spells have a requirement that the actor casting a spell must know how to cast it and be in a possession of a wand, as well as it targets.

8. Dementor

Cannot be an instance of HPActor since it needs extra functionality, HPActor lacks this required functionality, therefore it will be made into a subclass of HPActor and it will override some methods to perform its functionality. It doesn't relate to Patrol class since its movement is random.

Dementors have the following behaviour:

• All Dementors are on team EVIL. • A Dementor has a home base, which is its initial location. • At the start of its turn (a turn is when its act() method is called), if a Dementor is in a location with one or more actors on a different team, it sucks energy out of the actors, reducing each actor's hitpoints by 40. It then does the movement behaviour described below. • When a Dementor is at its home base, it waits for a random number of turns between 1 and 5 inclusive. It then randomly chooses a direction in which to travel, and randomly chooses how many steps to travel in that direction, up to a maximum of three steps. After that, on each turn it moves one step in its chosen direction until it cannot move further or it has moved the chosen number of steps. It then retraces its steps back to its base one step per turn.

9. Apparate

Extends Spell

The apparate spell would work fairly similar to actors are initialised and placed into the harry potter world except there are additional requirements, such as possessing a wand and knowing how to cast apparate. Like all other spells where there was a cast on a target actor option, there will now be an option to cast apparate. Then, the user will be prompted to enter in two coordinates that are valid locations on the map, and be moved accordingly via the entitymanager in hpworld. To check if a location is empty or not, it would check via the HPWorld for entities in a given location, and search for entities. In order to implement the splinching mechanic, as the actor is only travelling in a randomly chosen direction, a random direction will be generated using an RNG, and the actor will be moved via the move affordance until they are at an empty grid location.

10. Teach.

Extends HPAffordance implements HPActionInterface
All Actors can be taught if they are being set the teach affordance.
To check if the actor is a teacher the following aspects are checked for:

- it knows spells
- the attribute isTeacher is set to TRUE

When the HPActor meets anyone (HPActor) from its Team, it offers to teach.
The user is being asked if he wants to learn anything.
If the user has accepted, Teach class compares known spells from teacher and the HPActor and offers the list of spells that can be taught. User selects and the Player learns a chosen spell by adding it to its array list.

Specifically, only options to be prompted to the target for spells to be taught is spells that the teacher knows that the student doesn’t. i.e. if the targeted actor already knows a certain spell, it cannot and does need to be taught how to cast that specific spell again.

11. Secret Tunnel

For Assignment 3, a new Tunnel must be implemented, that 'runs' beneath the original grid.

There are 2 Doors that are each located in both the Tunnel and the original map, thereby linking the two grids.

Doors can be entered, which requires a new type of Affordance to be created.

There also needs to be a way to render only one Grid per loop in the main Application code, depending on the Player's location.

11.1. Tunnel

A special static boolean attribute inTunnel is added to the Player to specify if the Player is in tunnel.

If inTunnel is true it sets the grid to the tunnelGrid, otherwise it sets it to the myGrid (which is the base Grid).

A new Tunnel class is required to be created, with its own HPGrid.

It represents a hidden tunnel and will contain two Doors, which will link it to the main HPGrid.

Likewise, the main HPGrid will also have two Doors that are linked to the Tunnel's HPGrid.

The Tunnel itself is not aware of the link - the Doors do the linking. This maintains Doors re-usability.

11.2. Doors & Enter Affordance

A Door class extends HPEntity.

A Door is an Entity which stores Locations of where it leads to and from.

Door is given Enter Affordance

Enter Affordance will provide the Player an option to Enter the Door to get to the specified location in the destination grid.

11.3. Rendering the Right Grid

To render the right grid, the HPGridController needs to be capable of switching the HPGrid assigned to it.

A new method in HPGridController needs to be created - setGrid().

A new method in HPGridTextInterface also needs to be made, to allow it to render the right grid. Hence it also has a setGrid() method.

Upon querying the HPWorld's static chooseEntitymanager() where the Player is, the Grid assigned to the HPGridController will be switched accordingly which will then allow the HPGridController to render the right Grid during Application runtime

12. Inventory

An actor with `INVENTORY` capability allows the actor to hold up to three (3) items. This could include a wand, a potion, or some other collectable artefact, including two items of the same class. In previous implementation, actors could only hold up to one item at a time. This was achieved inside the `HPActor` class using the methods `getItem()` and `setItem()` and the item was stored in a private attribute `HPEntityInterface itemCarried`. This implementation lacked scalability in case an actor was to carry more than one item. To make actors hold up to three items (or any other positive number), the two previous methods were replaced with three new public methods and the items are store in a protected attribute `ArrayList<HPEntityInterface> Inventory` which its size is dictated by another protected attribute `int InventorySize.`

1. `void addToInventory(HPEntityInterface item)`
2. `void removeFromInventory(HPEntityInterface item)`
3. `ArrayList<HPEntityInterface> getItemsCarried()`

   In addition, 4 new public methods were created to decrease the reliance on `getItemsCarried()` for uses such as checking if the actor carries an item or if actors inventory is full. Therefore, theses specific method will decrease duplicated code as their functionality are used throughout the Harry Potter game.

4. `ArrayList<HPEntityInterface>getItemsWithCapability(Capability capability)`
5. `HPEntityInterface getHighestItemWithCapability(Capability capability)`
6. `boolean inventoryNotFull()`
7. `boolean carriesItems()`

   Actors that do not have the `INVENTORY` capability can still make use of all the new public methods above, with the only difference being `InventorySize` is set to one.

   Because these new methods share similar responsibilities, they were moved to a **new class called** `Inventory`. Whenever an HPActor is initiates, it initiates an inventory object.

   There is a one to one association between HPActor and Inventory classes. Thus for every HPActor instance there is one Inventory Instance.

   Note: When **Expelliarmus** is cast, the actor will drop a random item that has either a capability of `CASTING` or `WEAPON.` If the actor has items of the same capability, then the actor will drop the one with the highest hit points.

13. Broomstick

In possession of a Broomstick an actor can travel at double its usual speed. Broomsticks can only be acquired from a teacher after learning 2 spells. At any given time, an actor can possess one broomstick only. A broomstick is a subclass HPentity, it has a capability of `DOUBLESPEED`. DoubleMove action Allows the actor to travel at double speed when it detects the actor is carrying an item with `DOUBLESPEED` capability.

14. DoubleMove

If an actor possesses a Broomstick, then this actor can travel at double its usual speed. This is achieved by executing DoubleMove action instead of Move action. Because it uses most of the methods in the Move Class, DoubleMove is a subclass of Move class. It only overrides three methods

1. `canDo(HPActor a)`: same as the one in Move Class with one additional condition being if and only if the actor possess a Broomstick.
2. `act(HPActor a)`: executes if `world.canDoubleMove(a, whichDirection) returns ture`. Then it executes `world.moveEntity(a, whichDirection)` twice to perform its functionality correctly.
3. `getDescription()`: different description.

WORK BREAKDOWN ACKNOWLEDGEMENT

I, MOHAMED SHAKEEL MOHAMED RAFI [05/02/2019: 22:00], accept this breakdown arrangement.

I, MATTI HADDAD [05/02/2019: 22:00], accept this breakdown arrangement.

I, KERRY YUE SONG ZHENG [05/02/2019: 22:00], accept this breakdown arrangement.